METHOD AND APPARATUS FOR RETAINING ERROR-CONTROL CODE PROTECTION ACROSS BLOCK-SIZE DISCONTINUITIES

FIELD OF THE INVENTION

[01] This invention relates to error detection and correction in data that is being transferred from one location to another.

BACKGROUND OF THE INVENTION

[02] In applications involving the transfer of data from one location to another, it is frequently necessary to make changes to the data format along the way. The data headers and the size of the packets or blocks into which the data is assembled, in particular, may be dictated by such things as memory organization and address boundaries on the one hand and by the applicable communication protocol on the other. The result is that the "natural" block size at the data source and that at the data destination may be incompatible. Consequently, the error-control coding technique used to monitor the integrity of the data at its source cannot be used to protect it all the way to its destination.

[03] The prior-art technique for addressing this problem is to check the integrity of the data at each point at which the format has to be changed and then to re-encode the data using a coding technique that is compatible with the new format. This technique leaves the data unprotected during this transition; that is, there is generally no way of ensuring that the data has not changed between the integrity check and the calculation of the new code check.

[04] An even more serious problem resulting from the need to check the validity of an error-control code each time the data format changes is the concomitant increase in the complexity of the software that is typically involved in shepherding the data from its source to its destination. If the error-control code has to be checked each time the format changes, and, if that check indicates a problem, remedial action has to be taken. This requirement greatly complicates the software, increasing significantly the time needed to develop and test it, reducing its reliability and increasing its run time.

[05] If, in contrast, the error-control code needs be checked only at the destination, the intermediate code checks and diagnostic and recovery software can be eliminated from the main thread of execution. If a code fault is detected at the destination, the associated data block or packet is simply rejected and a separate software routine, external to the main execution thread, is used to diagnose the problem. Since data errors are generally rare and, in the vast majority of cases, are due to transient or intermittent events that are especially difficult to deal with in software, the benefit in avoiding the added performance and reliability costs of having multiple checks for such events is significant.

[06] The conventional solution to the problem of monitoring the integrity of data across format discontinuities is shown in Figure 1A. Data emerges from the data source 100 and is protected by an error-detecting, or an error-correcting, code. Typically, the codes used are systematic codes; that is, the data is transferred without modification followed by a code check calculated from that data and designed to expose any errors suffered by the data during the transfer. When the data with its appended code check is received at a destination, a verifier 102 regenerates the code check from the data and compares the regenerated code check to the code check received along with the data. If the two code checks are identical, the data is accepted as is. If not, the data is either rejected or corrected, depending on the type of code being used. To protect the data as it is being passed on to its destination (e.g., to a storage medium), the data is then repartitioned into blocks with an appropriate block size by a reformatter 104 and a new code check is calculated for each block by a code check generator 106 and appended to the new data block.

[07] Figure 1B is a flowchart of the steps needed to implement a representative example of such a procedure. Here, data is received and its integrity checked in step 108. A decision is made in step 110 whether the data passes the integrity check. If the integrity check fails in step 110, a diagnostic and recovery routine is initiated in step 126 in an attempt to identify the problem and take remedial action (e.g., inform the data source that the data was corrupted in transmission).

[08] Alternatively, if the data passes the integrity check as determined in step 110, the data is then reformatted and re-encoded in step 112 for transmission to a data cache where its integrity is again checked in step 114. If, as determined in step 116, the integrity check fails, another diagnostic and recovery routine is initiated in step 128 which is necessarily different from the previous diagnostic routine initiated in step 126 and which results in different remedial action.

[09] Alternatively, if the data passes this second test as determined in step 116, the data is then reformatted and re-encoded a second time in step 118 and passed on to its next destination, in this case a storage unit, where its integrity is checked once again in step 120. A Failure as determined in step 122 forces a third diagnostic and recovery routine to be called in step 130. This third diagnostic routine is specific to the new data format and code and results in yet another remedial response.

[10] Alternatively, if as determined in step 122, as will be true in the vast majority of cases, all the data is found to be valid at each integrity check, it is finally stored in step 124. Still, because of the rare event that a data error could be experienced during any of these transfers, separate diagnostic routines initiated in steps 126, 128 and 130, each adding complexity and introducing potential bugs to the main thread of execution, are required at each step.

[11] Therefore, there is a need for an apparatus and method that can protect data as the data is being reformatted without introducing a large amount of complexity.

SUMMARY OF THE INVENTION

[12] In accordance with the principles of the present invention, at each format discontinuity, the original data is broken into new data blocks and a code check is calculated from, and combined with, each new data block, but the new data blocks and new code checks are both reconstituted versions of the original data blocks and the original code checks. Consequently, the data is never left without protection.

[13] In one embodiment, an ingress encoder recomputes an ingress code check from an original data block and its associated header using the same algorithm that was used to originally compute the code check. An egress encoder computes an

egress code check using the same code check algorithm from the egress header for an outgoing data block reformatted from the original data block and the data portion of ingress code check. The outgoing information is then assembled from the egress header, the outgoing data block and the newly computed egress code check.

[14] In another embodiment, a controller/aligner subtracts the portion of the ingress code check that was generated from the ingress header from the contents of the egress encoder during the computation of the egress code check in order to remove the effects of the ingress header. The controller/aligner then adds the ingress code check to the contents of the egress decoder during computation of the egress code check.

[15] In yet another embodiment, the controller/aligner adjusts the ingress code check before it is added to the contents of the egress decoder to account for non-data bits or bytes added to the incoming data to align the data to word boundaries.

[16] In still another embodiment, the controller/aligner rotates the ingress code check before it is added to the contents of the egress decoder to account for non-data bits or bytes added to the incoming data to align the data to word boundaries.

[17] In yet another embodiment, the ingress header and data are modified before the ingress code check is computed to account for non-data bits or bytes added to the incoming data to align the data to word boundaries.

[18] In another embodiment, the ingress and egress encoders are arithmetic encoders that generate a one's-complement sum of the ingress and egress data to be encoded.

[19] In still another embodiment, the ingress and egress encoders are encoders that generate a vertical-parity check code from the ingress and egress data to be encoded.

[20] In another embodiment, the ingress and egress encoders are cyclic-residue code encoders that generate a cyclic residue code from the ingress and egress data to be encoded.

BRIEF DESCRIPTION OF THE DRAWINGS

[21] The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings in which:

[22] Figure 1A is a block schematic diagram illustrating a prior art apparatus for protecting data with code checks.

[23] Figure 1B is a flowchart illustrating the steps needed to implement the prior art method of protecting data with code checks.

[24] Figure 2A is a block schematic diagram illustrating an apparatus for protecting data with code checks constructed in accordance with the principles of the invention.

[25] Figure 2B is a flowchart illustrating the steps needed to implement the inventive method of protecting data with code checks.

[26] Figure 3A is a block schematic diagram of a conventional arithmetic code check encoder for encoding a serial data stream.

[27] Figure 3B is a block schematic diagram of a conventional vertical-parity-check code check encoder for encoding a serial data stream.

[28] Figure 3C is a block schematic diagram of a conventional cyclic-residue-code check encoder for encoding a serial data stream

[29] Figure 4A is a block schematic diagram of a conventional arithmetic code check encoder for encoding a parallel data stream.

[30] Figure 4B is a block schematic diagram of a conventional vertical-parity-check code check encoder for encoding a parallel data stream.

[31] Figure 4C is a block schematic diagram of a conventional cyclic-residue-code check encoder for encoding a parallel data stream

[32] Figure 5 is a block schematic diagram of an encoding apparatus constructed in accordance with the principles of the invention.

[33] Figures 6A and 6B, when placed together, form a flowchart illustrating the steps in a method for operating the apparatus of Figure 5.

DETAILED DESCRIPTION

[34] In accordance with the principles of the present invention, as illustrated in Figure 2A, code checks are calculated from data generated by data source 200 and appended to the data as in the prior art, but the new data blocks and new code checks are both reconstituted versions of the old data and code checks as calculated by the verifier/formatter 202. The data is never left without protection.

[35] A flow chart depicting the data processing at the data format discontinuities set forth in the previous example is now greatly simplified as shown in Figure 2B. The data is now simply received in step 204, reformatted, and transferred to the buffer cache in step 206. Code checks are recalculated in step 206 using the same coding algorithm that was originally used to calculate code checks for the data received in step 204. The data and recalculated code checks are sent on to the next stage where they are received in step 208 and reformatted in step 210. Again, the code checks are recalculated using the same algorithm as used in step 206.

[36] Data integrity is checked only at its final destination in step 212. A determination is made in step 214 whether the integrity check has been passed. If found to be valid, then the data is stored in step 216. Otherwise, the process proceeds to step 218 where the data packets from which the storage block is composed are rejected and a message to that effect is sent to the source. Only one diagnostic routine is needed; it is called from the main program thread but run in a background mode in an attempt to determine the source of the problem.

[37] Since the data has been rejected at this point, no attempt need be made to try to recover it. Should the problem be the result of a transient event of some kind, as it is in the large majority of cases, operation will continue normally, without interruption. If, instead, the problem is due to a permanent hardware malfunction, subsequent data packets will also be found to be invalid. In this case, since permanent faults are generally easy to isolate, the background diagnostic routine should quickly identify the problem.

[38] The present invention can be used in conjunction with any of a large class of error-control codes involving sequential operations on successive data elements.

Typical codes used for this purpose include arithmetic codes, vertical-parity codes and cyclic-residue codes (CRCs). An arithmetic code of the type used for Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) communication is illustrated in Figure 3A. In order to generate the code from binary input data, the data is shifted into an *n*-bit shift-register comprised of stages 300-306. When each group of *n* bits, called a "word", have been shifted in, the contents of the register are added to the, initially all zero, contents of an adder, also comprised of adder elements 310-316. Each adder element calculates the binary sum of the data input, its current contents and the carry input from its neighbor, as indicated by the arrows, to produce its new contents (which it retains) and carry bit. The adder thus contains the one's-complement sum of all the words shifted into the shift-register since the adder was last reset. This sum, or, in the case of TCP and UDP, the one's complement of this sum, is then used as the code check.

[39] A vertical parity check encoder is shown in Figure 3B. It differs from the previous coding technique only in the way in which the successive words are operated on to form the code check. In this case, the contents of a register comprising stages 330-336 are combined, using exclusive-OR gates 340-346, with the corresponding bits of each new data word after the new data word has been shifted into the input shift-register stages 320-326. Thus, at any instance, the *k*th register stage (of stages 330-336 contains the exclusive-OR of the *k*th bit of every word that has been presented to it since it was last reset. The contents of the register stages 330-336 after all data words in the block have been received constitute the code check.

[40] Cyclic-residue codes are also used to protect data transferred over certain communication links, for example Ethernet and Fibre Channel links. An example of a CRC generator is shown in Figure 3C. Here, the shift-register itself, comprised of stages 350-356, constitutes the generator. As new data bits are shifted in, the register contents are modified by forming the exclusive-OR, using a subset of the exclusive-OR gates 360-368, of the bits that may be shifted into some of its stages with the bit in its last stage (356). The bits that are shifted into the stages are determined by which of the connections, symbolically illustrated by the switches 370-376 are actually implemented.

After all the data that is to be encoded is presented to the shift-register, which is initially set to a known value (usually all zeros) its contents define the code check.

[41] For purposes of the subsequent discussion, it is useful to recognize that if the data is interpreted as a binary polynomial $d(x)$ of degree $N-1$, with N the number of bits to be encoded, and the feedback connections are used to define the coefficients of a binary polynomial $g(x)$ of degree n, then the CRC code check is just the value of $d(x)$ modulo $g(x)$. That is, if $d(x) = d_0x^{N-1} + d_1x^{N-2} + \ldots + d_{N-2}x + d_{N-1}$, with d_k the kth data bit and if $g(x) = x^n + g_{n-1}x^{n-1} + g_{n-2}x^{n-2} + \ldots + g_1x + 1$, with $g_j = 1$ if and only if the feedback connection to the ith stage ($0 \leq i \leq n - 1$) of the shift-register shown in Figure 3C is in fact made, then the code check is equal to $d(x)$ modulo $g(x)$.

[42] The encoders in Figures 3A-3C are designed to accept data serially. This is generally the case when the data is received over a communication link. Sometimes, for example, when data is being read out of a random-access memory, it is more natural to present it to the encoder in parallel, one word at a time. The encoders of Figures 3A-3C are shown in Figures 4A-4C modified to accept parallel inputs. Both the one's-complement and the vertical-parity-check encoders are actually simplified with this change since the serial-to-parallel converting shift-registers needed for the serial implementation can be eliminated here.

[43] In the one's-complement encoder shown in Figure 4A, each bit of an n-bit word is entered into an adder comprised of adder elements 410-416. Each adder element calculates the binary sum of the data input, its current contents and the carry input from its neighbor, as indicated by the arrows, to produce its new contents (which it retains) and carry bit. The adder thus contains the one's-complement sum of the word entered into the adder elements. This sum, or, in the case of TCP and UDP, the one's complement of this sum, is then used as the code check.

[44] The vertical parity check encoder is shown in Figure 4B. The input bits are applied as one input to exclusive-OR gates 450-456. The other inputs to the exclusive-OR gates 450-456 are the contents of the associated register stages 440-446. The output of each exclusive-OR gate is stored in its corresponding register stage.

Thus, at any instance, the *k*th register stage contains the exclusive-OR of the *k*th bit of every word that has been presented to it since it was last reset. The contents of the register stages 440-446 after all data words in the block have been received constitute the code check.

[45] The parallel CRC encoder becomes somewhat more complicated, however, with the simple exclusive-or gates (360-368) in the serial case replaced by a more complex exclusive-or logic tree 480 as indicated in Figure 4C. Logic tree 480 generates the same function of its contents that would be obtained by shifting the register in Figure 3C *n* times, with *n* the length of the register. For example, let $n = 4$ and let the contents of the last stage be fed back into the first and second stages (so that $g(x) = x^4 + x + 1$). Then, as is easily verified, the contents of the first through the fourth stages of the shift-register after four shifts are $x_0 + x_3$, $x_0 + x_2 + x_3$, $x_1 + x_2$, $x_0 + x_1$, respectively, with x_0, x_1, x_2 and x_3 the initial contents of those stages, and the exclusive-OR logic is defined accordingly. (The symbol "+" here indicates addition over the binary field, i.e., the exclusive-OR operation.)

[46] An important feature of the coding techniques described in the previous paragraphs is that they are all "linear" in the sense that the sum of valid code checks on any two blocks of data is itself a valid code check on the sum of the data blocks. The meaning of "sum" here depends on the way in which the code check is defined. For the previously described arithmetic code, sum means the one's complement sum. For the vertical-parity-check and CRC codes, sum means the module-two sum (exclusive-OR) of corresponding bits. The CRC code differs from the other two, however, in the way in which two blocks have to be aligned for the linearity property to hold. With the one's-complement and vertical parity codes, only word alignment is required. That is, the sum of the code checks on two data blocks is a valid code check on the sum of the two blocks regardless of how the words comprising those blocks are ordered or of which words are summed with which other words. The same statement is true for CRC codes only if the codes are block aligned and the corresponding bits are summed. The term "block" in this context means the shorter of (1) the block of data that is being encoded and (2) the length of the sequence generated by the CRC generator polynomial. Since

the former is generally considerably shorter than the latter, block-alignment will usually mean alignment of the two data blocks themselves.

[47] In the following discussion, the terms "code check" and "code" will be used interchangeably when there is no danger of ambiguity. The "inverse" y of an entity x will mean its additive inverse, i.e., the quantity such that $x + y = 0$ with "+" indicating "sum" as previously defined. Specifically, if sum is the one's complement sum, the inverse of x is its one's complement and if sum is the modulo-two sum, the inverse is just x itself.

[48] In addition, when zero-stuffing is used to align two data streams, the term "alignment" in this context will mean alignment, modulo the word length, when the code check is defined by either the one's complement or the vertical-parity-check codes and alignment, modulo the block length, when the code check is defined by a CRC.

[49] The terms "word oriented" and "block oriented" will be used to distinguish codes whose linearity properties depend, respectively, on word and block alignment.

[50] The encoding procedure of the present invention is illustrated in Figure 5. The modifier "ingress" is used there to denote the information (header and data) as received from the source and "egress" is used for the information that is to be sent to the destination after an appropriate format modification. The two encoders shown in Figure 5 (500 and 504) can be either serial or parallel versions of any one of the three types of encoders shown in Figures 3A-3C and 4A-4C. Generally, serial encoders are used when the ingress data is serial in nature (e.g., when received over a communication link) and parallel encoders are used when it is inherently parallel (e.g., when it is retrieved from random access memory or received over a parallel bus). Note that the code itself can be serially extracted from a parallel encoder and unloaded in parallel from a serial encoder, so it can be appended to the egress information in whichever form is more convenient.

[51] Ingress data including any header information is re-encoded, as it is received, as indicated by arrow 512, in encoder 500 using the same code check algorithm that was originally used to encode the data. When an incoming block of data has been received in its entirety, the code generated on that block in encoder 500 is

compared to the code received with the data in the comparator 508. If the two codes agree, the data is accepted; if they disagree, an error is flagged.

[52] Since the ingress and egress headers are generally different, the egress information consists of the catenation of the egress header 516 with the data portion of the ingress information as indicated schematically by arrow 514. The multiplexer 506 is used to concatenate the new header 516 with the ingress data 514 to produce the egress information 518.

[53] The egress encoder 504, in contrast to the ingress encoder 500, does not encode the data directly; rather it generates code checks for the egress information by combining its own intermediate contents with inputs from the ingress encoder 500, via the controller/aligner 502 and from the egress header indicated by arrow 510. The aligner function 502 is needed to compensate for the fact that, in some cases, the ingress data will not be properly aligned with its egress counterpart so the ingress code generated by encoder 500 must be re-aligned accordingly. For expository purposes, it is initially assumed in the following description that the ingress and egress data are properly aligned. The steps required to compensate for misalignment will be addressed subsequently.

[54] The sequence of operations needed to convert a valid code on the ingress information to a valid code on the egress information without actually re-encoding the data are illustrated in the flowchart shown in Figure 6. In step 602, the egress encoder 504 is used to generate the code for egress header using the same code check algorithm that is used by the ingress encoder. This can be done in parallel with the use of the ingress encoder 500 to encode the ingress header and data as set forth in step 604. The encoding process in step 604 continues until it is interrupted by any one of three events: the end of an ingress header (EOH), the end of an ingress data block (EOB) possibly coincident with the simultaneous end of an egress data block, or the end of an egress data block not coincident with the end of an ingress data block. The existence of one of these events is checked in step 606. If no event is detected, the process returns to step 604 to continue the encoding process.

[55] If an end-of-ingress-header event occurs the process proceeds to step 605. At this point, the controller/aligner 502 causes the inverse of the contents of the ingress encoder 500 to be added to the contents of the egress encoder 504. This addition cancels out the contribution of the ingress header when, as will be seen shortly, the ingress code generated by the encoder 500 over both the ingress header and the ingress data is added to the contents of the egress encoder 504. The process then returns to step 604.

[56] Next, in step 604, data is sequentially presented to ingress encoder 500 until another event is recognized in step 606 that interrupts this process. If a block-oriented code is being used, one *zero* binary bit or one *all-zeros* binary word is presented to the egress encoder 504 during this data phase for each bit or word presented to the ingress encoder 500. This is done because the contribution of a bit or word to the code depends on where it appears in the block. Presenting zeros to the egress encoder 504 while data is being presented to the ingress encoder 500 maintains proper block alignment and ensures the correct contribution of each bit or word to the final code.

[57] Alternatively, if, in step 606, an end-of-ingress-data-block event is detected, the process proceeds instead to step 612. Since a complete ingress header and data block has been received at this point, the received and calculated ingress codes are now compared by comparator 508; if they disagree, an error is flagged and an error recovery routine (not shown) is executed. Otherwise, the contents of the ingress encoder 500 are added to the contents of the egress encoder 504 and the ingress encoder 500 is then reset. Since the effect of the ingress header has already been subtracted out, the net contribution from the ingress encoder 500 is just that of the data portion of the ingress information. In addition, since the contribution to the code of the egress header has also been previously added in, the egress encoder 504 at this point contains a valid code on the egress header concatenated with the ingress data. The process then proceeds to step 614.

[58] If no more ingress data is to be included in the egress data block as determined in step 614, the process proceeds to step 616 where zeros are appended to

the data if necessary to complete the egress block and, in the case of block-oriented codes, zeroes used as inputs to the egress encoder 504. The egress encoder 504, at the conclusion of this process, then contains a valid code on the egress header and data and can be used to monitor the integrity of that data as it is transferred and stored elsewhere. The code is therefore appended to the egress block and both the ingress encode 500 and the egress encoder 504 are reset.

[59] However, if the just completed ingress block was not the last block to be included in the egress block, as determined in step 614, the process returns to step 604. The next ingress block is presumably preceded by another header. That header is encoded in step 604 as before and the process continues, first using the ingress encoder 500 to generate the code on the next ingress header, as before, and then continuing to encode the ingress data.

[60] If the end-of-egress-data-block event is detected in step 606 before all the ingress data has been reformatted, the process proceeds to step 610. In step 610, the current contents of the ingress encoder 500 are added to the contents of the egress encoder 504 to form the egress code. Again, since the header contributions to the ingress information have already been subtracted out, the net contribution to the egress code is just that from the portion of the ingress data that has been included in the egress data block. The egress encoder therefore contains a valid code on the egress header and data. Once this code is transferred out, the egress encoder 504 is reset and used to calculate the code on the header of the next egress block and the inverse of the code in the ingress encoder is added to the result. This last step negates the effect that the data included in the current egress block has on the ingress code when it is used to construct the code for the next egress block.

[61] In some cases in which word-oriented codes are used, the data block does not necessarily end at a word boundary. For example, with TCP and UDP, the word length is sixteen bits, but the data portion of the transmission may consist of an odd number of bytes. When this happens, the last byte in the last transmitted word is set to all zeros. When several ingress blocks are packed into a single egress block, these filler bits have to be deleted since they do not represent data.

[62] To compensate for this, the code generated on the next ingress block has to be rotated by an amount equal to the number of non-data bits appended to the previous block. This correctly compensates for the deletion of these non-data bits in the egress block provided the code remains valid under rotation; that is, so long as the code on a set of rotated data words is an equal rotation of the original code on the non-rotated words. It is easily seen that this property is true for both the one's-complement and the vertical-parity-check codes.

[63] To perform this compensation, the aligner 502 in Figure 5 is set to rotate the next code transferred from the ingress encoder 500 to the egress decoder 504 by the appropriate amount at the end of steps 610 and 612. Alternatively, the aligner 502 could be placed at the input to the ingress encoder 500, thereby shifting the header and data bits before they are encoded rather than the code bits afterwards.

[64] Since it is generally true that both the ingress and egress headers are word aligned (i.e., the header ends and the data starts on a word boundary when word-oriented codes are used), no special alignment procedures are usually needed at the end of steps 602 and 608; that is, the alignment is left unchanged at those points in the procedure. If this is not the case, alignment adjustments can be made at the end of those steps as well.

[65] Since block-oriented codes do not have a word substructure, alignment is generally not an issue when codes of this type are used. If fixed-length blocks are used, however, and bits that have been stuffed into the ingress data stream to complete a block are removed from the egress data stream, an alignment procedure is required for these codes as well. In this case, rather than realigning the ingress code, it is only necessary to inhibit the shifting of zeros into the egress encoder 504 in step 604 for a number of times equal to the number of bits that were stuffed into the ingress data stream. (If a parallel encoder of the type shown in Figure 4Cc is used as an egress encoder 504, it has to be modified to support shifts equal to the number of stuffed bits if that number is not an integral multiple of the encoder length. This can be easily done by anyone knowledgeable in CRC encoder design.)

[66] Although hardware encoders are implied in the preceding discussion, they are by no means required. The entire procedure can be readily accomplished in software. The modulo-two addition required for the CRC and the vertical-parity-check codes is easily implemented in a general-purpose processor and the addition needed for the one's-complement code is only slightly more difficult. The rotation required for the alignment of word-oriented codes is also easily done in software. The shift-register operations needed in step 604 to achieve the alignment needed to calculate a valid egress code when CRCs are used, however, are potentially considerably more cumbersome without hardware assistance. Even these operations can be easily accomplished, however, using a general-purpose processor and a relatively small random-access memory.

[67] For example, assume a k-bit data stream represented by $d_k(x)$ is to be concatenated with an m-bit data stream represented by $d_m(x)$ and that the code is defined as the residue of this concatenation modulo $g(x)$. Then, if $r_k(x)$ is the residue of $d_k(x)$, modulo $g(x)$, and $r_m(x)$ is the residue of $d_m(x)$, modulo $g(x)$, the residue of the concatenated data stream, represented by $x^m\, d_k(x) + d_m(x)$, is $x^m\, r_k(x) \bmod g(x) + r_m(x)$. But $x^m\, r_k(x) \bmod g(x)$ is equal to the residue modulo $g(x)$ of the quantity $x^m \bmod g(x)$ multiplied by $r_k(x)$. The operation $x^m\, r_k(x) \bmod g(x)$ can therefore be implemented using a random-access memory consisting of m words, each of n bits, with $n - 1$ the degree of the generator polynomial $g(x)$, by storing a word representing the residue $\rho_i(x) = x^i \bmod g(x)$ at location i for all i in the range $(1,m)$, multiplying that polynomial by $r_k(x)$ and then using the same memory to determine residues of the at most $n - 1$ terms in the resulting polynomial of degree $2n - 2$ or less having exponents exceeding $n - 1$. The final step is simply adding, modulo two, the corresponding coefficients of those memory look-ups to each other and to the corresponding coefficients of $r_m(x)$. The whole operation is thus accomplished with at most n memory accesses and modulo-two, n-bit additions and one binary polynomial multiplication. Since m is limited by the maximum block length of interest, the memory needed for this purpose can be relatively small. If the ingress block length is constrained to be, say, an integral number of bytes, then the number of

possible values of *m* is further reduced by a factor of eight and the memory can be correspondingly smaller.

[68] A software implementation of the above-described embodiment may comprise a series of computer instructions either fixed on a tangible medium, such as a computer readable medium, e.g. a diskette, a CD-ROM, a ROM memory, or a fixed disk, or transmissible to a computer system, via a modem or other interface device over a medium. The medium either can be a tangible medium, including, but not limited to, optical or analog communications lines, or may be implemented with wireless techniques, including but not limited to microwave, infrared or other transmission techniques. It may also be the Internet. The series of computer instructions embodies all or part of the functionality previously described herein with respect to the invention. Those skilled in the art will appreciate that such computer instructions can be written in a number of programming languages for use with many computer architectures or operating systems. Further, such instructions may be stored using any memory technology, present or future, including, but not limited to, semiconductor, magnetic, optical or other memory devices, or transmitted using any communications technology, present or future, including but not limited to optical, infrared, microwave, or other transmission technologies. It is contemplated that such a computer program product may be distributed as removable media with accompanying printed or electronic documentation, e.g., shrink wrapped software, pre-loaded with a computer system, e.g., on system ROM or fixed disk, or distributed from a server or electronic bulletin board over a network, e.g., the Internet or World Wide Web.

[69] Although an exemplary embodiment of the invention has been disclosed, it will be apparent to those skilled in the art that various changes and modifications can be made which will achieve some of the advantages of the invention without departing from the spirit and scope of the invention. For example, it will be obvious to those reasonably skilled in the art that, although the description was directed to a particular hardware and/or software system, other hardware and software could be used in the same manner as that described. Other aspects, such as the specific instructions utilized to

achieve a particular function, as well as other modifications to the inventive concept are intended to be covered by the appended claims.

[70] What is claimed is: